NIST Smart Flow System Programmer's guide

Christelle Martin
Olivier Galibert, Martial Michel, Fabrice Mougin, Vincent Stanford

http://www.nist.gov/smartspace/
National Institute of Standards and Technology, USA
Version 3.7, Last updated: December 13, 2000

Contact us:
Vincent Stanford - Vincent.Stanford@nist.gov
N.I.S.T.
Bldg 220, Rm A231
100 Bureau Drive, Stop 8940
Gaithersburg, MD 20899-8940 - USA

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The NIST Smart Space project

The NIST Smart Space Laboratory has been created to offer assistance to industrial research and product development laboratories as to face the numerous performance and interoperability challenges inherent in the Smart Work Spaces of the future. We believe that the shift to Pervasive Computing is already well under way and will have as much impact on industry, and daily life as personal computing did. Our Modular Test Bed is designed to allow our industrial and academic laboratories to bring their technologies together for integration and performance testing needed.

Pervasive Computing refers to the trend to numerous, easily accessible computing devices connected to each other and to an ubiquitous network infrastructure. This will create new opportunities and challenges for IT companies as they place computers and sensors in devices, appliances, and equipment in buildings, homes, workplaces, and factories. Within a few years, embedded devices able to execute complex software applications, and use wireless communications will be the norm and their effective use requires distributed interfaces on numerous, small, and even invisible devices.

We are prototyping an experimental smart space data-flow system. The prototype focus is on advanced forms of human-computer-interaction. Integrating wireless networks with dynamic service discovery, automatic device configuration, and sensor based perceptual interfaces. The objectives are to facilitate:

- Identify security mechanisms needed to ensure privacy, integrity, and accessibility of implementations
- Develop metrics, test methods, and standard reference data sets to pull the technology forward
- Provide reference implementations to serve as models for possible commercial implementations
- Interconnect the prototype components and systems to explore key issues associated with distributed smart spaces
- Establish an integrated multi-sensor perceptual interface test bed for smart work spaces
- Support integration of the AirJava/Aroma components
- Deploy perceptual interface components from our industrial advisors to investigate system level performance and metrics issues
- Provide a multi-sensor data recording environment for the production of standard test materials

We have developed the Smart Space Modular Test bed which consists of a defined middle-ware API for real-time data transport, and a connection broker server for sensor data sources and processing data sinks. For example, a microphone array acquires a speech signal, reduces it to a single channel, and offers it as a data-flow. Then a speaker identification system subscribes to the signal flow, while a speaker-dependent speech recognition system subscribes as well. The speaker ID system then offers a real-time flow, to which the speech recognition system also subscribes. This layer makes it possible if not necessary easy to integrate components that were not intentionally designed to work together, such as speaker identification, and speech recognition systems. Many constituent technologies are under separate development in industry, so the issues of interoperability and integration are paramount. This project will develop metrics and reference material based on real implementations for industrial use. Important technologies include is sensor-based collaborative interfaces using:

- Speech recognition
- Speaker identification
- Face recognition
- Source localization/separation
- Channel normalization
- Immersive video and acoustic displays
- Personal information appliances
- Pervasive networking
- Information storage and retrieval
- multimedia data types
- Multiple Interconnected Spaces

To foster integration, interoperability, and the development of emerging technologies, standardization and measurements are critical but for economic reasons, often are not addressed by individual companies. The lack of common software infrastructure, tools to create, manage, measure, test, and debug pervasive services, standards in key areas such as service discovery, APIs, wireless networks, automatic configuration, and ad hoc transactional security, all currently impede widespread adoption of pervasive computing. Also, measurements, and tests developed and performed in a public forum allow competing research systems to be compared, and improved systems to build on the best features of previous iterations.

It is a long-term research platform that will provide a sensor-rich collaborative working environment. Sensor technologies include microphone arrays, video camera arrays, smart badges, I.R. room scanners, and possibly person position sensors. The major component areas will include:

1.2 About this document

In this document, we introduce you to NIST Smart Flow System and help you getting started. The programmer's guide explains the system workings, so that programmers know how to suppress, add, modify objects (functions, types, etc.) from the platform.

The communication architecture has been implemented for the Linux Operating System, using the C/C++ programming languages. For this reason the User's guide is accessible to anyone having a basic programming background. However, the Programmer's guide requires knowledge in data processing communication system.

Chapter 2

Conceptual view

2.1 Introduction

The Pervasive Computing environments of tomorrow will require the efficient transmission of sensor data among a variety of heterogenous clients and servers. As such, an efficient data management and transmission protocol system will be vital.

The NIST Smart Flow System (NSFS) is an initial effort in developing such a system.

The NIST SDFS manages data flow among clients and data servers. The servers and clients are linked via a crossbar switch. Clients access data streaming from the servers via ”Flows” and ”Flow objects”. They communicate with the flows using the `SFlib` object access library.

2.2 Servers

Servers are completely connected via a crossbar switch.
Communication among them is accomplished via three object types:

- Client objects (CO)
- Flow objects (FO)
- Control center (CC)

Figure 2.1 shows as an example a NSFS with five local servers.

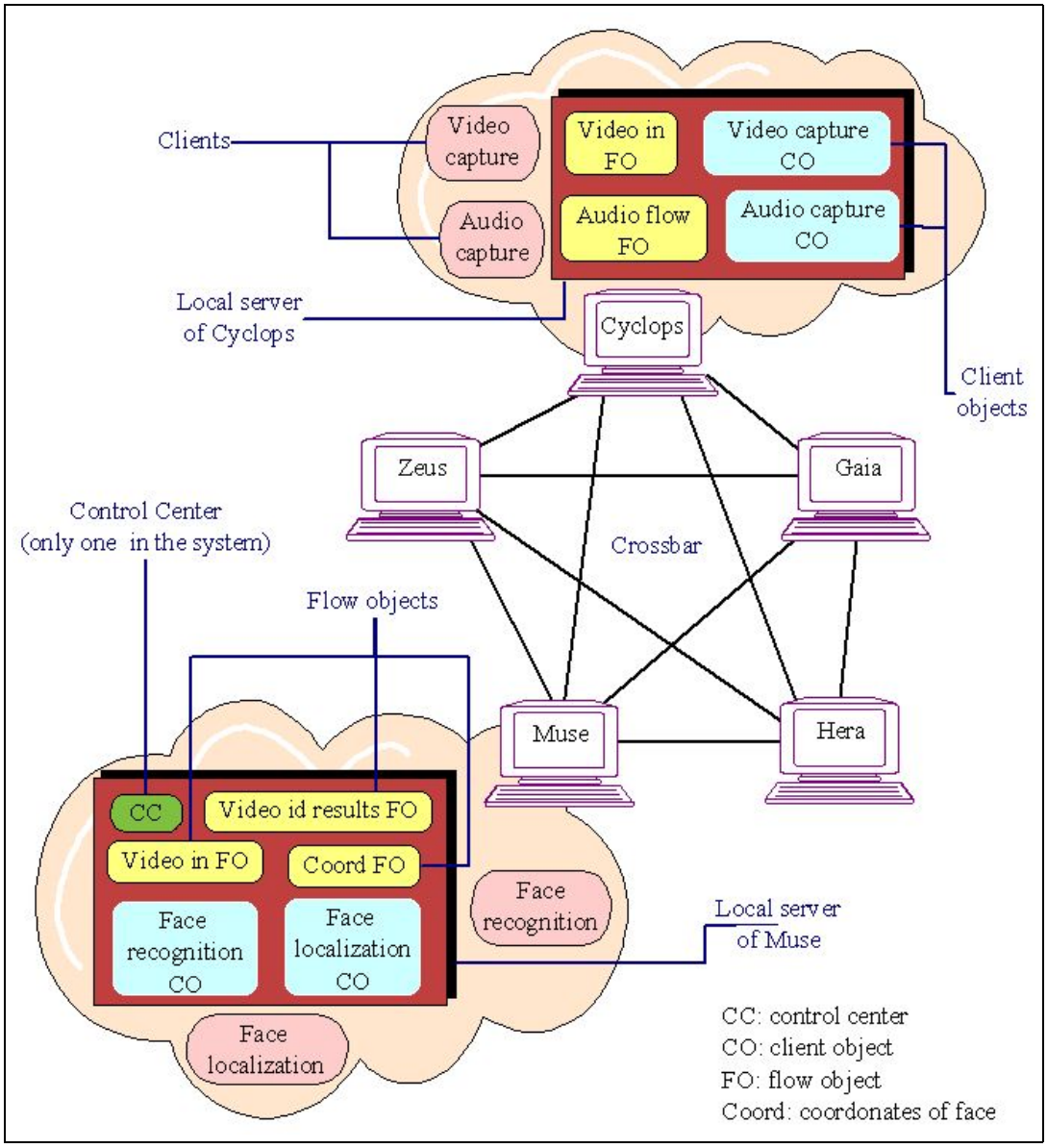

Figure 2.1: The communication system

2.3 Clients objects

2.3.1 Role

A client object (CO) receives requests from the real client it represents, and processes them. To illustrate the communication process, consider the `Face localization` process in figure 2.1. In this example, the `Face localization` client sends requests to the `Face localization` client object.

2.3.2 Existence

There is only one client object per actual client.
There can be several clients per machine.
A client using a flow doesn't know whether another client using the same flow runs on the same machine that it does or not.

2.3.3 Meaning

The server sees a client object as the representative of a connected client.

2.4 Flow objects

2.4.1 Role

Flow objects (FOs) manage flow communications via local clients that are using them.

2.4.2 Existence

When a server uses a certain flow, it accesses it via a specific flow object.
A server that doesn't use a certain flow has no flow object for it.
Every flow in the system has at least one associated flow object.

2.4.3 Meaning

A flow is an abstract entity that exists within a flow network.
A flow object is what a local server sees of a flow: Muse's `Video in` flow object contains everything that this server knows about `Video in` flow.

2.4.4 Example

To illustrate the role of a flow object, figure 2.2 explains how `Video capture` client sends a picture to `Face identification` client, assuming that connections are already established.

1. `Video capture` sends a message to `Video capture` client object via the `SFlib` library, to inform the local server that it has a new image.
2. `Video capture` client object then looks for the local `Video in` flow object and informs it that a new image is available.
3. This flow object sends this picture to all flow objects subscribers to `Video in` flow.
4. `Video in` flow object of `Face localization` local server receives the image and tells `Face localization` client that it has a new image.

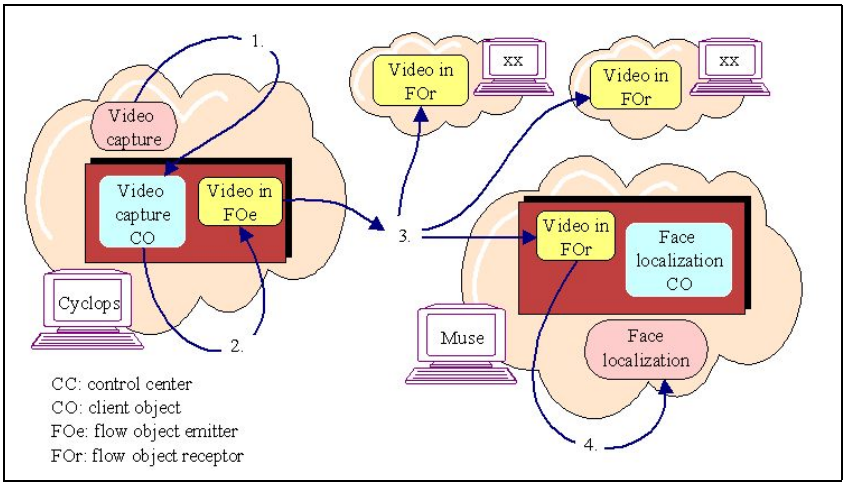

Figure 2.2: Data sending process when connections are established.

2.5 Control Center

2.5.1 Role

The control center (CC) carries out the connections among flows. It contains the list of the flows, subscriptions, components, and all meta-information relative to flows.

Every object is able to send a message to the control center.

2.5.2 Existence and Meaning

This particular object is a "centralizer", there is one and only one control center in the system. It can be on any server of the system, nothing is pre-required.

2.5.3 Example

As an example, we will see what happens when the creation of a new connection is requested. This situation can appear in two circumstances:

- A client is asking for a data on a flow that doesn't exist on its machine.
- A client wants to send a data on a flow that doesn't exist on its machine.

Section 2.6 on page 10 gives as an example the illustration of this second case: Cyclops wants to open `Video capture` flow in emit mode.

To simplify, this process is decomposed in two parts.

First part

As explained in the diagram figure 2.3, here is the first part of the process occurring when a new connection is requested:

- The client contacts its client objects.
- The CO sees that there is no FO corresponding to the required flow. Thus, it creates a local flow object, let's say for example LFO.
- LFO informs the control center about its creation and sends name, group and type of this new local flow to the CC.
  - If this flow doesn't already exist:

    The CC sends back to LFO an identification number for the new flow.
  - If this flow already exists:
    - If there is a type-conflict between them:

      The CC refuses the creation and sends LFO the error message "`type mismatch`". The new FO forwards this error message to the client and self-destructs.
    - If there is no type-conflict:

      Cf. the second part.

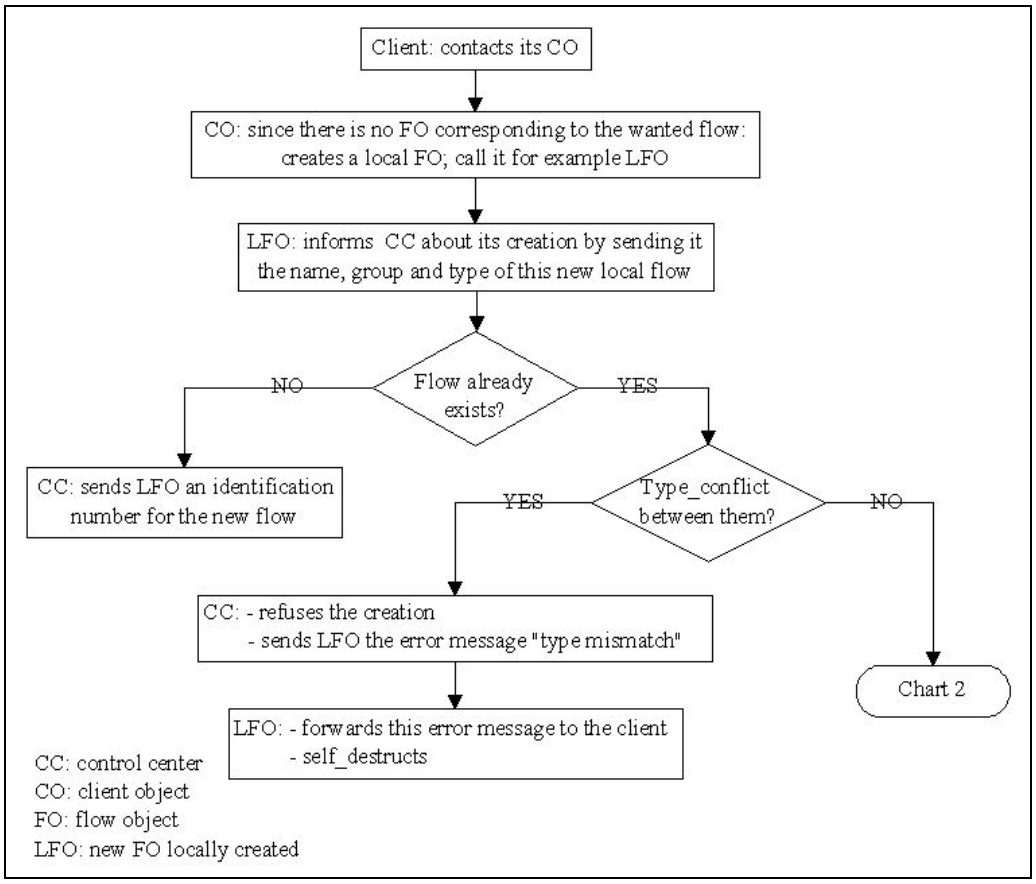

Figure 2.3: Role of the control center in a new connection request: Chart 1

Second part

The diagram on figure 2.4 explains the second part of the creation process of a new connection. That is when the requested flow already exists and when there is no type-conflict between this existing flow and the new local one.

- If it is a request to emit whereas there is already a emitter for this flow (Remember that a flow is from one to n.):

  The CC refuses to create a new flow and send LFO the error message "`flow already created`". LFO self-destructs after having forwarded the error message to the concerned client.

- If it is a request to emit whereas there is no emitter for this flow:

  The CC sends LFO the identification number of the existing flow.

  LFO sends the CC an acknowledge receipt after receiving the identification number.

  The CC can then inform all flow objects subscribers to this flow of the existence and location of the emitter.

  So, they inform the emitter that they are interested in this flow.

Thereupon, the emitter sends all those FOs the following message:
"OK, the number of the first buffer is..., the type of the data is (ex.: jpeg for video, frequency or number of bits for audio)..."

- If LFO is a receptor:
  - If no emitter exists at this moment:

    The CC just sends LFO the identification number of the existing flow.
  - If there is an emitter for this flow:

    The CC sends LFO the identification number of the flow and the location of the emitter.
    LFO can inform the FO emitter that it is interested in the flow.
    Then it receives the number of the first buffer and the type of the data from the emitter.

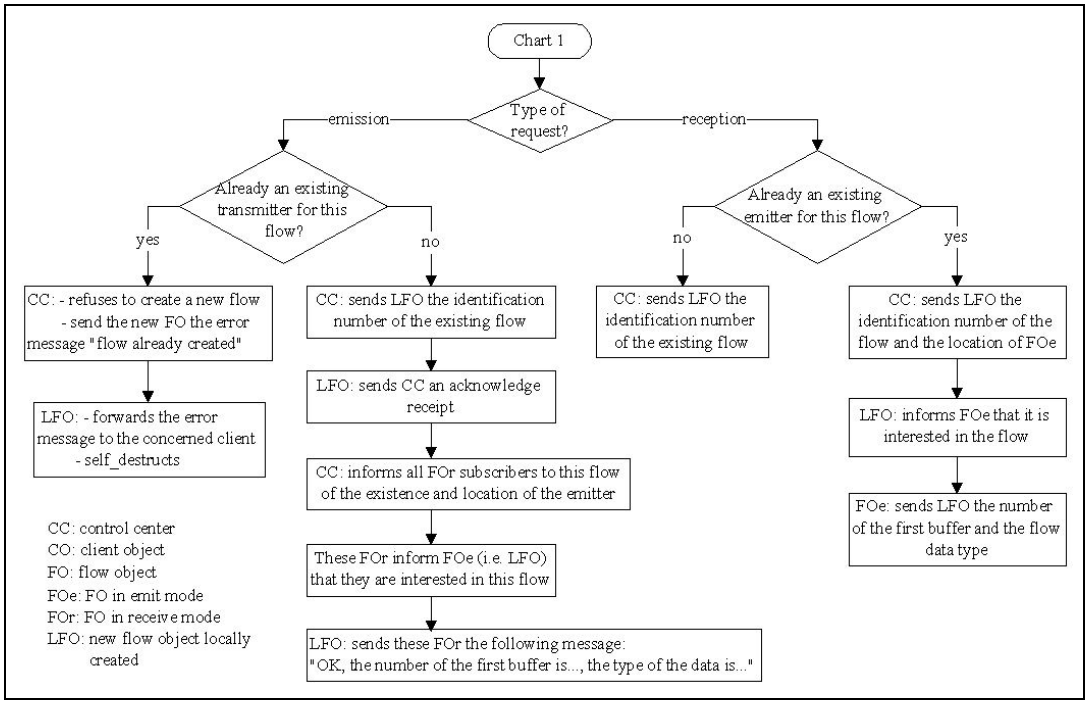

Figure 2.4: Role of the control center in a new connection request: Chart 2

2.6 Example: How to open a flow in emit mode

To illustrate the role of these objects, the process of opening a flow in emit mode is step by step described below.

2.6.1 Initial situation

The initial situation consists in a client that wants to emit on a flow that doesn't exist on its local server. As described in figure 2.5, this corresponds to:

- In `Camera 1` group `Video capture`, a client of Cyclops, wants to send data.
- This information has to be emitted on `Video in`, a data flow that already exists.
- At this point, there is no emitter for this flow.
- Cyclops hasn't used this data flow i.e., it has no `Video in` flow object.

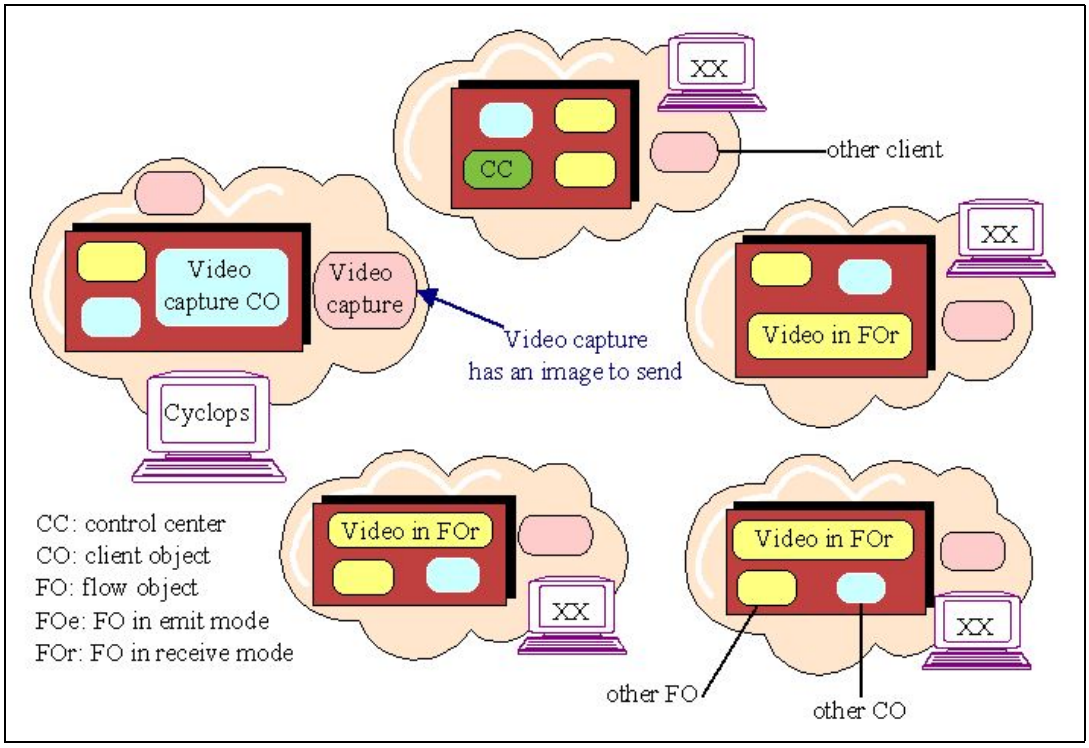

Figure 2.5: Opening a flow in emit mode: initial situation

2.6.2 Creation of a `Video in` flow object on the local server of Cyclops

Then Cyclops' `Video capture` CO creates a local `Video in` FO on its local server, and the CC accepts this creation. The steps are described in figure 2.6:

1. `Video capture` contacts its client object.
2. `Video capture` CO creates a local `Video in` FO since it can't find an existing FO corresponding to the requested flow.
3. This local FO informs the CC of its creation, sending it the name, group and data type of the new local flow.
4. The CC sends Cyclops' new `Video in` FO the identification number of the existing flow since
   - `Video in` flow already exists,
   - there is no type-conflict between the new local flow and the corresponding existing one,
   - it is a request to emit and there is no emitter for the existing `Video in` flow.

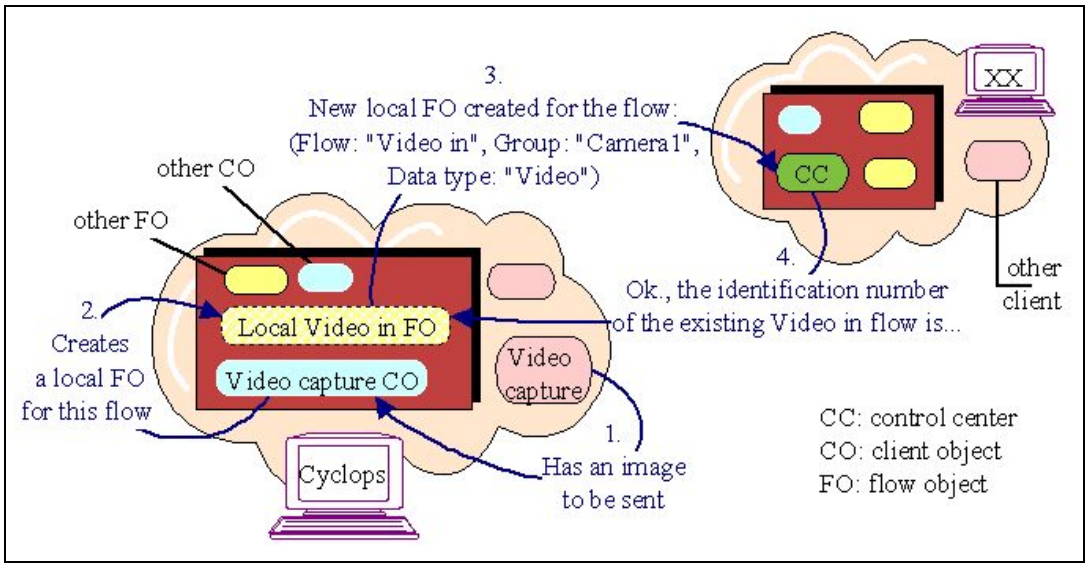

Figure 2.6: Opening a flow in emit mode: creation of a local flow object

2.6.3 Informing the other `Video in` flow objects

As shown in figure figure 2.7:

5. The new `Video in` FO sends back the CC an acknowledgement.
6. The CC then tells all FOs subscribers to `Video in` flow that there is now an emitter for this flow and sends them its location.

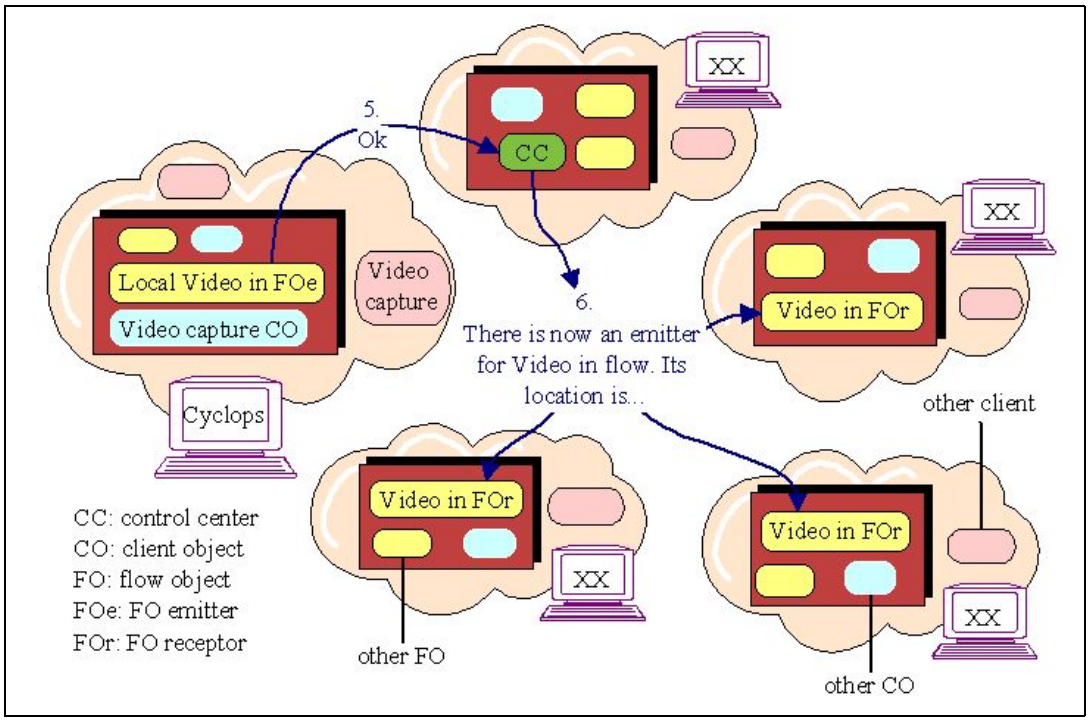

Figure 2.7: Opening a flow in emit mode: the control center informs the other `Video in` flow object

2.6.4 Flow objects receptors make themselves known to the flow object emitter

Then, as figure 2.8 illustrates it:

7. The FOs subscribers to `Video in` flow inform the new FO emitter that they exist and give it their locations.

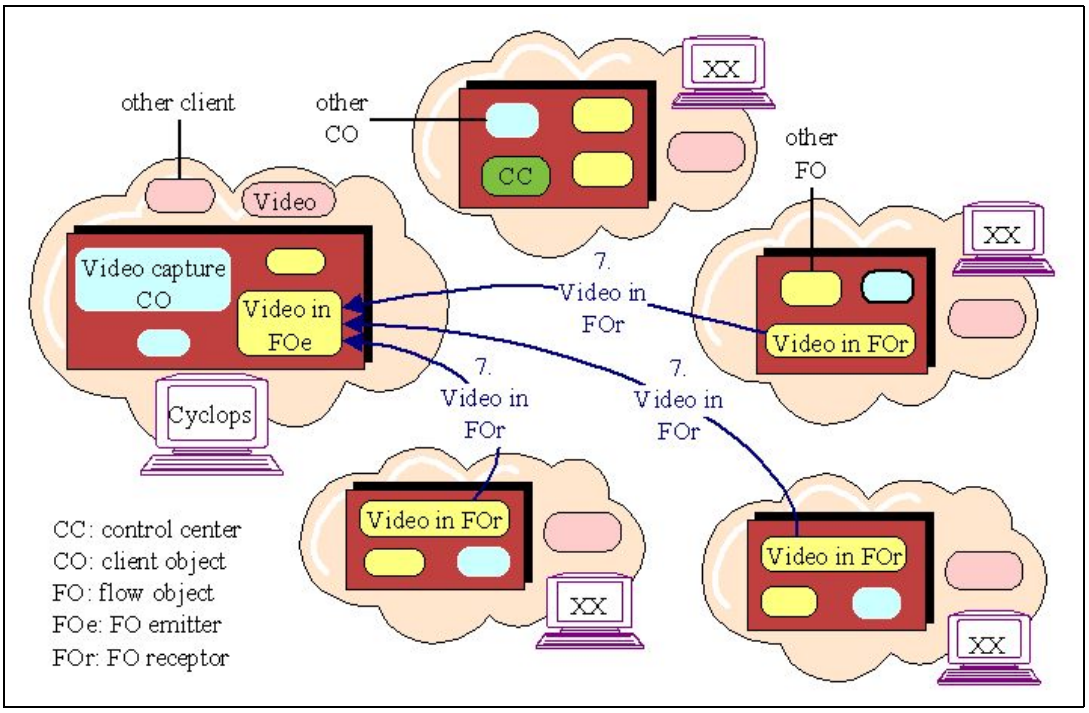

Figure 2.8: Opening a flow in emit mode: `Video in` flow objects inform the flow new emitter of their existence and location

2.6.5 "Video capture" sends data

Finally, as shown in figure 2.9:

8. Cyclops' `Video in` FO sends an acknowledgement to all `Video in` FOs receptors, and gives them the number of the first buffer and the data type.

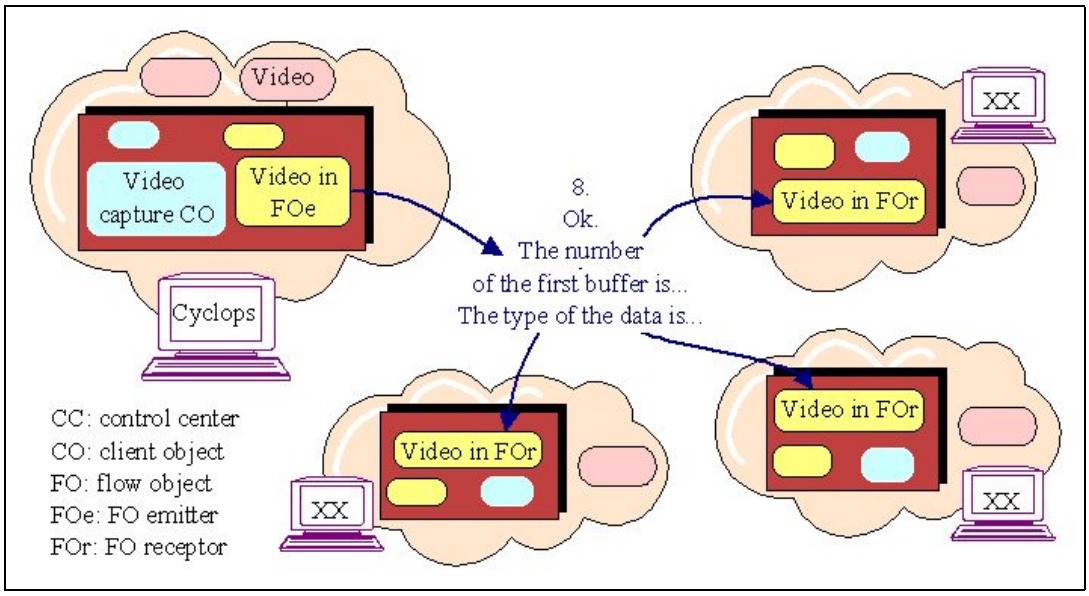

Figure 2.9: Opening a flow in emit mode: `Video capture` sends data

Chapter 3

Connections between clients and their local server

Communication is done using:

- Unix sockets,
- Shared memory,
- Semaphores.

3.1 Unix sockets

Unix sockets are bidirectional pipes in which clients can write data in a specified order and read them in the same order. This bidirectional communication is system-local (ie., you can not have a Unix socket running on the network.).

Unix sockets are used to send control messages, requests and replies. However, sockets are required to accommodate wait states and blocking. Therefore, in practice this method is not efficient for transmitting time critical streaming data.

3.2 Shared memory and semaphores

For this reason, data are sent through a shared memory. Its access is synchronized with semaphores.

Flow data are stored in the shared memory, and their existence is independent from the clients that access them. For instance, a flow object systematically posts the data upon a request. That way a client subscriber to this flow can access the data in the shared memory without re-contacting its client object.

The main idea here is time saving: any buffer that is still in the shared memory can be accessed without re-contacting the server.

3.3 Representation of the shared memory

When a client opens a flow in receive or emit mode, its local server gives it the following information:

- the flow header position in the shared memory (`sh_flow_header`)
- the position of a shared memory block, specific to the subscription (`sh_client_info`)
- the buffer header position in the shared memory (`sh_buffer_list_header`)

This chapter gives the description of these three elements and illustrates them figure 3.1.

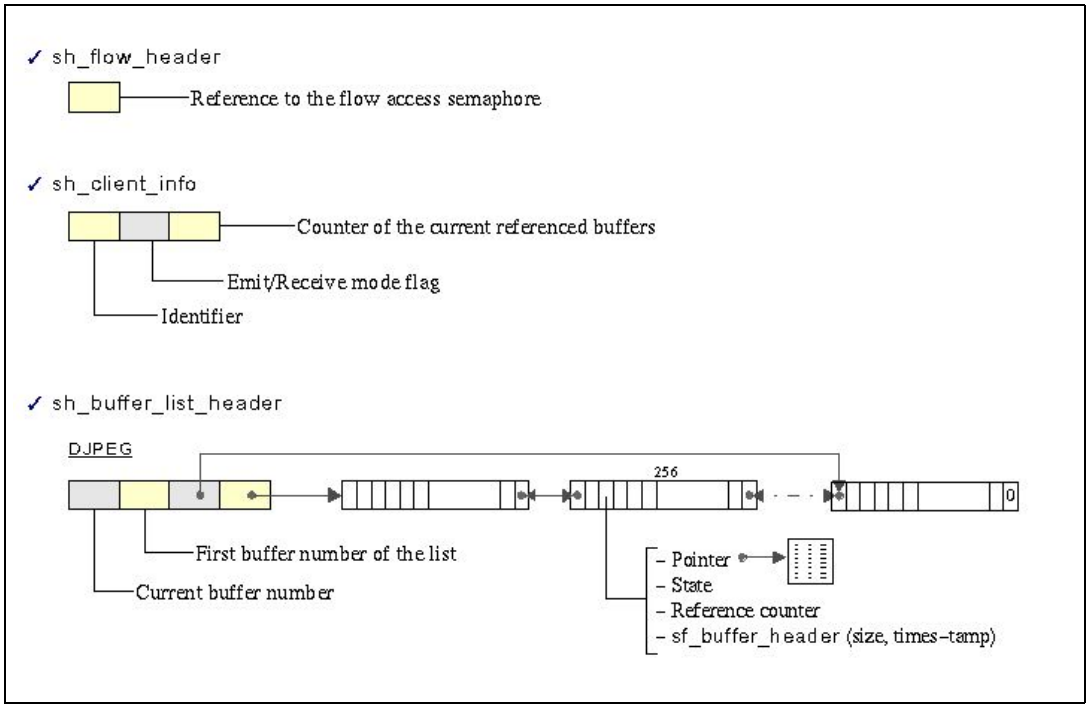

Figure 3.1: Representation of the shared memory

3.3.1 sh_flow_header

`sh_flow_header` keeps a reference to the semaphore that controls access to the flow data.

3.3.2 sh_client_info

`sh_client_info` holds:

- an identifier, to know which type of data it is associated with when communicating with the server
- a flag for the subscription mode: emit or receive
- a counter of the number of buffers currently referenced by the client (This is a safety device: if a client crashes, we can know how many buffers it had.)

3.3.3 sh_buffer_list_header

There is one buffer header per data type and thus, at least one buffer header, as shown in figure 3.1.

`sh_buffer_list_header` has:

- the current buffer number
- a pointer to a double linked buffer list
- the list first buffer number
- a pointer to (the first element of) the list last buffer

The buffer list in question is a list of 256 objects plus a pointer to the next list. Each buffer holds:

- a pointer
- a reference counter
- the state
- `sh_buffer_header`
- (and an internal pointer)

The pointer

It is a pointer to the buffer effective data (when those data exist).

The reference counter

The reference counter value is:

- equal to 0 when the buffer is not in use
- greater than 0 when the buffer is in use (it is then forbidden to destroy the buffer)

The state

As shown and explained in table 3.1, there are five possible states:

- `allocated`
- `unallocated`
- `present`
- `converting`
- `cached`

sh_buffer_header

The client gets a `sh_buffer_header` internally that has information about the buffer itself, i.e. size and time-stamp. Time-stamp is the exact buffer creation instant.

States	Meaning	Comment
`unallocated`	The buffer is currently not in use. (pointer = NULL)	This situation can occur in two cases: -1- The buffer has aged out of storage and its list hasn't been destroyed yet. -2- The buffer is "too far in the future" and has not yet been allocated.
`allocated`	The emitting client has space enough to store data.	The memory space has been allocated, the pointer is updated, but there is no data inside.
`present`	The buffer can be used.	Data has been written, everything is OK.
`converting`	The server is converting data into the wanted format.	The data is not usable yet.
`cached`	The buffer can be used.	This state is used for the multicast transport layer. It is equivalent to `present` for other prurposes.

Table 3.1: The five possible states of the buffer pointed by `sh_buffer_list_header.`

Chapter 4

Internal workings of a server

This chapter describes how servers work, in an object language (c++) viewpoint.

4.1 The core

The core of each server holds a dispatcher: a class called `Multiplexer class`. This class manages `Connections` and knows which ones are active. `Connections` are a group of objects siblings of the `Connection class`.

The `Multiplexer` class detects message arrivals coming from the server's clients or from other machines, and new connections of clients or machines. It informs the objects of any new message arrival.

4.2 Main Connections objects

The main `Connection objects`, subclasses of the `Connection class`, are:

- `Server object`
- `Client object`

4.2.1 Server object

The `Server object` (`Server class`) corresponds to a connection with another distant server. It is the one that communicates with the other servers.

4.2.2 Client object

The `Client object` (`Client class`) corresponds to a connection with a real client.

4.3 Other Connection objects

Besides these two main objects, you will find six others:

- `Control center`
- `CCEntry class`
- `LRequest`
- `CommGlobals`
- `Flow objects`
- `Flow manager`
- `Memory manager`

4.3.1 Control center

The location of the `control center` (`CCenter class`) in the system is given to servers when they are launched.

4.3.2 CEntry class

The effective implementation of this class is done by its two siblings:

- `CCLocal,`
- `CCRemote.`

There is only one `CCEntry` per server, appearing under one of these forms in the `CommGlobals`. Requests for the `control center` correspond to `CCEntry` methods calls. With this class, the code stays the same, whether the control center is local or not.

4.3.3 LRequest

A client creates a `LRequest` (i.e. Local Request) whenever it gets a new message - a request from another client via a flow object. This object contains the type of the message, its content and all information going with it, so that an answer can be sent to the client (i.e. a flow object can directly reply to the connected client.).

4.3.4 CommGlobals

`CommGlobals` keeps the list of:

- the identifiers,
- the number of clients with the corresponding objects,
- the number of flows with the corresponding objects,
- the number of servers,
- the number of requests.

A request is a triplet (`flow`, `client`, `LRequest`) to which a number is dynamically assigned. All of these fields don't necessarily exist. When a server communicates with the control center, this number is always given so that the context can be retrieved when getting an answer.

4.3.5 Flow objects

All `flow objects (Flow class)` are gathered together in another object: the flow manager. The `flow class` has three subclasses: `FlowAudio`, `FlowVideo` and `FlowData`.

4.3.6 Flow manager

The `flow manager (FlowManager class)` has the list of all `flow objects`, so they can be found whenever they are needed. Look-ups are made via this object.

4.3.7 Memory manager

The `Memory manager (MManager class)` manages the shared memory. It provides allocators when someone needs to use the shared memory.

Chapter 5

Conclusion

NIST Smart Flow System architecture has several great advantages. The structure is totally dynamic and the system is network transparent, i.e. independent from the network structure.

Great care has also been taken to reduce memory bandwidth requirements, and make good use of network bandwidth.

Nevertheless, we can't start several communication systems using the same computers at the same time. Actually, the system operates with peer-to-peer communication among the clients implemented with `SF_LIB`, and does not have closed boundaries for the Smart Flow application.

Since this is an evolutary architecture, this document will be updated in the future. Of course, any comment about this paper or the communication system is welcome.